deque

Building Accessible Android Apps with Jetpack Compose

What is Jetpack Compose?

Modern Android UI toolkit for building native Android UI faster and easier by using intuitive DECLARATIVE Kotlin APIs.

Which means no more xml or imperative type view building.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity"
    android:id="@+id/scroll"
    >
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/text2"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:text="Already have an account? Login"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        />
```

Imperative

```kotlin
@Composable
fun HeadingLarge(text: String, color: Color = Color.Black, modifier: Modifier = Modifier) {
    Text(
        text = text,
        fontSize = 30.sp,
        fontFamily = FontFamily.Cursive,
        textAlign = TextAlign.Center,
        fontWeight = FontWeight.Bold,
        color = color,
        modifier = modifier
    )
}
```

Declarative

Why Jetpack Compose?

- Faster and easier Android UI development(especially for developers like me!)
- Easy to use intuitive Kotlin APIs
- Easy to catch bugs in UI
- Less code(Who doesn’t like that?)
- Built-in support for Material Design, Dark Theme and Animation

Also…

- Gives you the ability to build custom UI

```
BasicTextField(value = , onValueChange = )
```

- Easy to create reusable UI components

```
@Composable
fun HeadingLarge(text: String, color: Color = Color.Black, modifier: Modifier = Modifier) {
    Text(
        text = text,
        fontSize = 30.sp,
        fontFamily = FontFamily.Cursive,
        textAlign = TextAlign.Center,
        fontWeight = FontWeight.Bold,
        color = color,
        modifier = modifier
    )
}
```

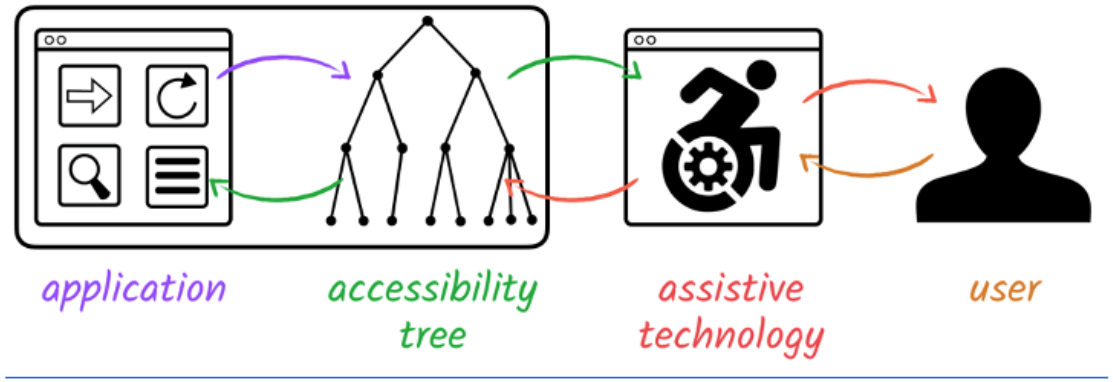
What about Accessibility?

Compose gives immense power to developers to build literally any type of UI without worrying about increasing code and maintainability but…

The responsibility of making our applications accessible for every user

How Accessibility Services work in Android?

- Accessibility Services like: Talkback or SwitchAccess
- It is a service when turned on runs in the background and responds to the Accessibility Events like click, focus, tap and hold, etc.
- The screen/window content is arranged in a tree and each node in the tree is represented as an AccessibilityNodeInfo.



Source: blog.intuit.com

- As the name suggests AccessibilityNodeInfo provides all the accessibility related info to the Accessibility Service for a particular node in the tree.
- Information like: Name, Role, value, Action.
- Android highly recommends that: “Accessibility services should only be used to assist users with disabilities in using Android devices and apps.”

How Imperative(or xml) type UI interacts with Accessibility Services in Android?

Imperative UI with Accessibility Service

For each and every view in the view hierarchy the accessibility service has to extract the information which is going to define its behavior with Accessibility Technology like Talkback or SwitchAccess.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity"
    android:id="@+id/scroll"
    >
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/text2"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:textSize="25sp"
        android:text="Already have an account? Login"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        />
```

Imperative type UI

For this text view an Accessibility Node Info will be created which will have following info:

- text: Already have an account login
- isImportantForAccessibility: true
- boundsInScreen, etc.

How Jetpack Compose interacts with Accessibility Services in Android?

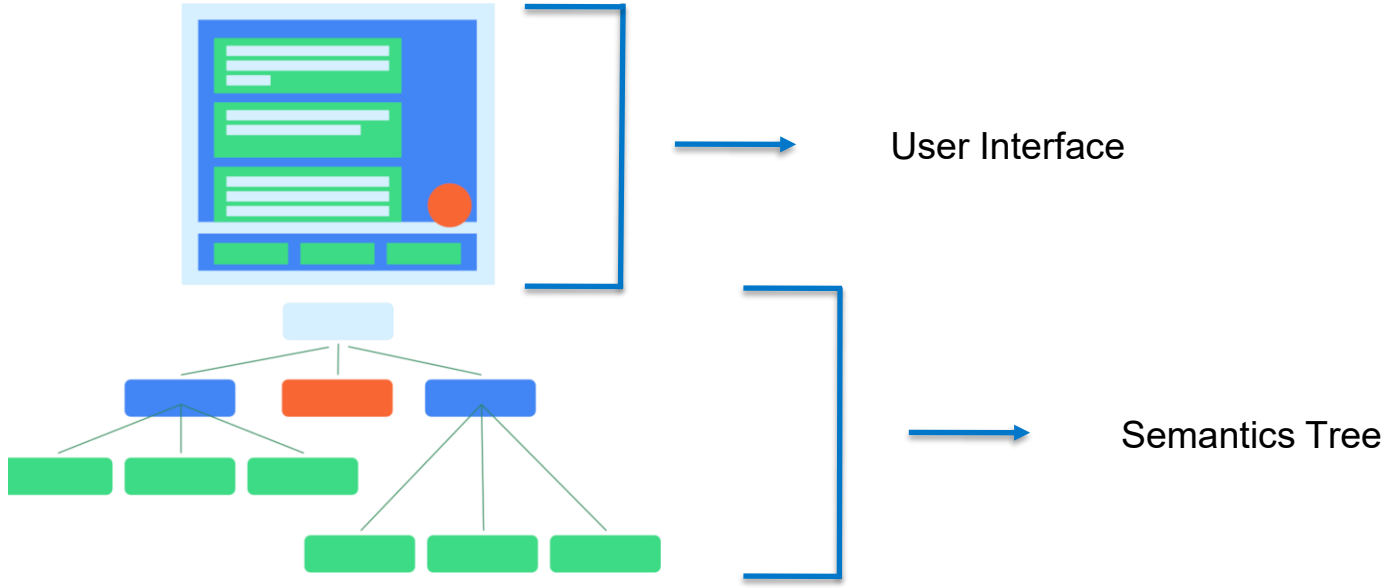
Semantics in Compose

Semantics are the information source for Accessibility services like Talkback or Switch Access to gather data from. It does NOT define information on how the composable will be drawn.

How Android Accessibility Services work with Compose?

Android creates a compose hierarchy tree which helps the accessibility service to figure out:

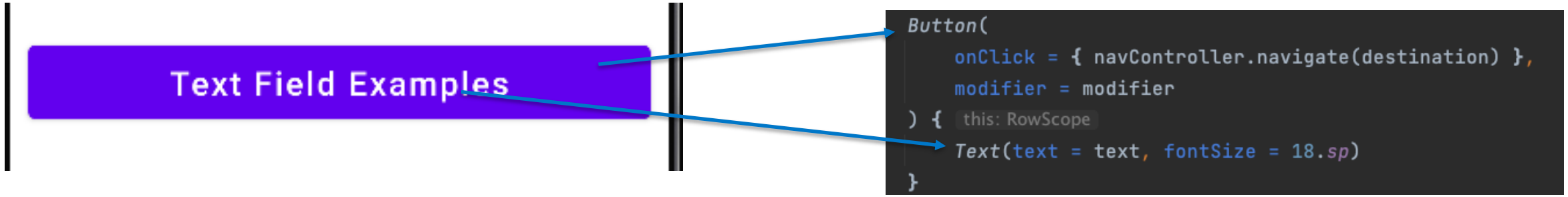
- The order of iteration from one composable to another.
- What to announce when a composable is focused or to focus a composable or not.
- What actions can be taken for a particular composable, for example: Click, check or custom accessibility actions(hmm!! Interesting..)
- Basically every information that AccessibilityNodeInfo requires to represent itself in the Tree.



Source: https://developer.android.com/jetpack/compose/semantics

Compose allows you to look at the composition hierarchy with just one line of code

Merged and Unmerged Composition Hierarchy example



Simple button in compose

Merged and Unmerged Composition Hierarchy example


```
|-Node #8 at (l=30.0, t=608.0, r=1050.0, b=729.0)px
| Role = 'Button'
| Text = '[Text Field Examples]'
| Actions = [OnClick, GetTextLayoutResult]
| MergeDescendants = 'true'
```

Merged Composition Hierarchy

```
|-Node #8 at (l=30.0, t=608.0, r=1050.0, b=729.0)px
| Role = 'Button'
| Actions = [OnClick]
| MergeDescendants = 'true'
|  |-Node #9 at (l=260.0, t=632.0, r=821.0, b=705.0)px
|  | Text = '[Text Field Examples]'
|  | Actions = [GetTextLayoutResult]
|  |-Node #1000000008 at (l=0.0, t=0.0, r=0.0, b=0.0)px
|    Role = 'Button'
```

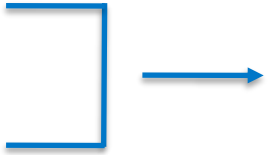
Unmerged Composition Hierarchy

Accessibility Service refers to the merged composition hierarchy
To gather information for a composable.

Composition hierarchy can be extracted while running the automated tests(we will see that later)

Enough talking, show me some code

```
Row(
    modifier = Modifier.border(width = 2.dp, color = Color.Black)
        .clickable(onClick = {
            Toast
                .makeText(context, text: "You have clicked on the item!", Toast.LENGTH_SHORT)
                .show()
        })
        .padding(16.dp)
        .semantics { this: SemanticsPropertyReceiver
            // Here I am telling the Accessibility Service
            // to treat this Row like a Button
            role = Role.Button
```



Using Modifier of a composable one can set semantics of a composable

In the above example “role” is one of the SemanticsPropertyKeys which sets the role of that row to Button so that when Talkback or SwitchAccess focuses on that row it will be announced as Button and hence action will be announced as “Double Tap to Activate”

There are many other useful Semantics Properties like:

- ContentDescription
- StateDescription: usually with switch(on/off) or checkbox(checked/unchecked)
- Heading
- Disabled
- IsPopup

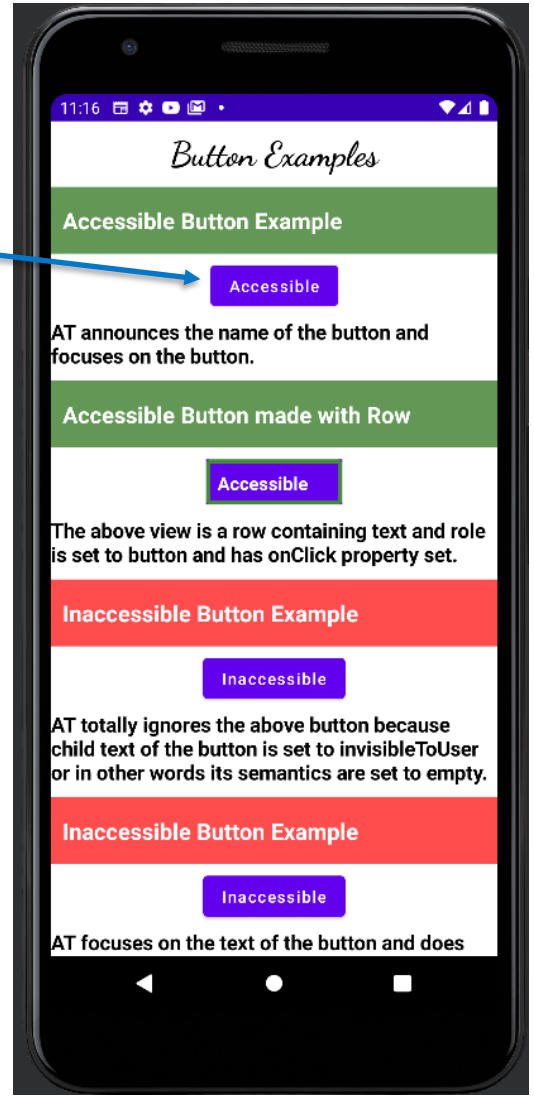
One does not have to add all the semantic properties one by one, most of them are inherited from the parent


MORE EXAMPLES
PLEASE

Buttons

The first button is accessible because talkback announces: “Accessible, Button. Double tap to activate.” So the
name: “Accessible”,
role: “Button” and
action: “Double Tap to activate”
of the button are announced by the Talkback. This how it looks like in code:

```
Button(
    onClick = { Toast.makeText(context, text: "This is an Accessible Button!", Toast.LENGTH_SHORT).show() }
    modifier = Modifier.constrainAs(example1) { this: ConstrainScope
        top.linkTo(divider1.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) { this: RowScope
    Text(text = "Accessible Button")
}
```



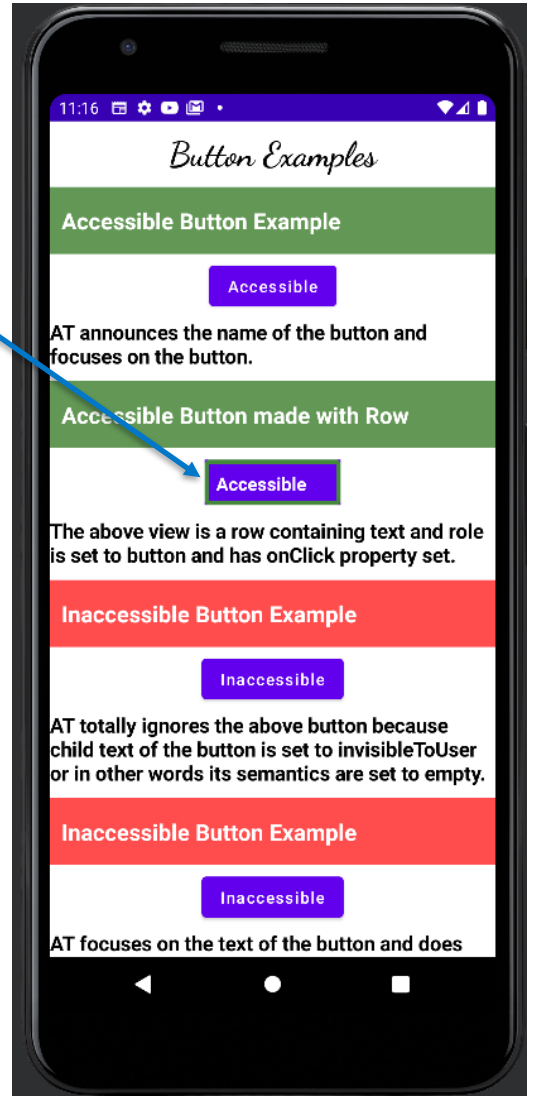
The “onClick” defines the action associated to the button and Actions are part of semantics similar to role.

Buttons

The second button is a simple row which is acting like a button but remember it is our responsibility to set the Role semantics property of the row to “Button”. The button is accessible because the name, role and action are announced by the Talkback. This is how that button looks like in code:

```
Row(
    modifier = Modifier
        .size(height = 40.dp, width = 120.dp)
        .background(color = Color( red: 98,  green: 0 ,  blue: 238))
        .constrainAs(example4) {  this: ConstrainScope
            top.linkTo(divider4.bottom, margin = 10.dp)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
        }
        .clickable(
            enabled = true,
            onClick = {
                Toast.makeText(context,  text: "Row as Button", Toast.LENGTH_SHORT).show()
            },
            role = Role.Button
        )
) {  this: RowScope
    Text(
        text = "Row Button",
        fontWeight = FontWeight.Bold,
        modifier = Modifier.padding(10.dp),
        color = Color.White
    )
}
```

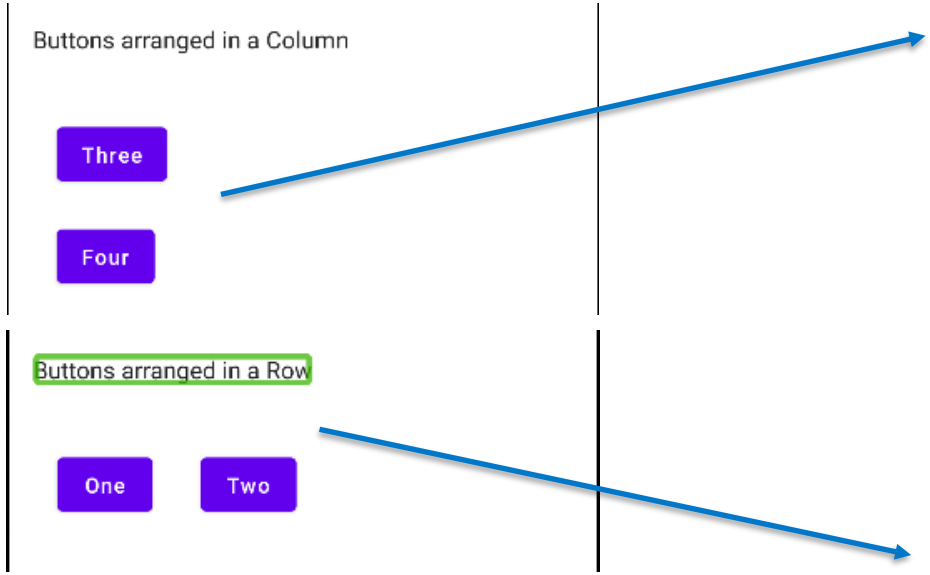
A Row in Jetpack Compose refers to a collection on Composables arranged one below another.
If the Row choose to merged the descendants of a row then the Accessibility Service treats it like One single element.



Talkback announces: “Accessible, Button. Double tap to activate.”

Rows and Columns

A Row is a layout composable that places its children in a horizontal sequence.
A Column is a layout composable that places its children in a vertical sequence.



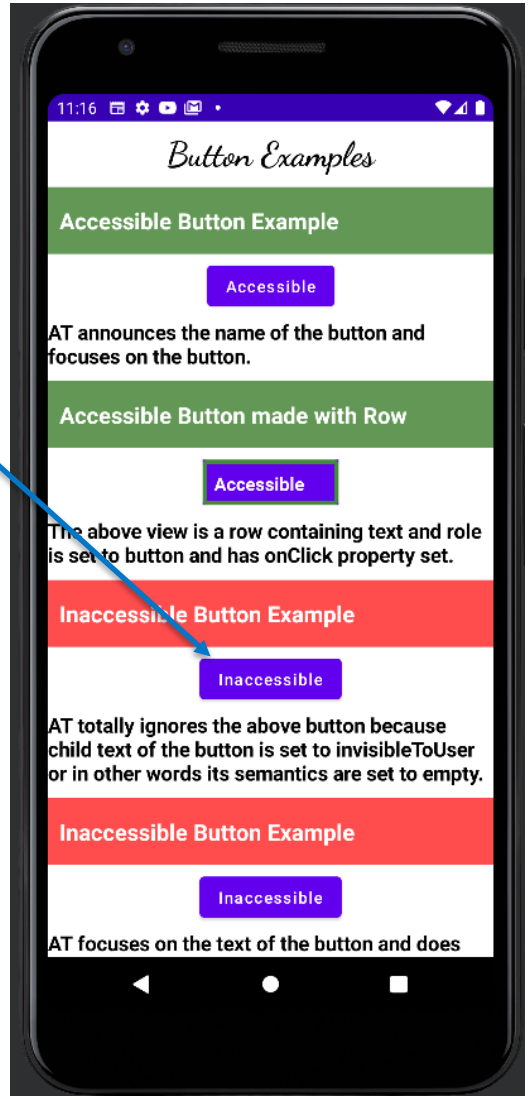
The buttons are arranged in a Column. That is one below another.
This helps talkback in determining the order of focus.

The buttons are arranged in a Row.
This helps talkback in determining the order of focus.

Buttons

If a composable is set to “invisibleToUser” the Accessibility Technology ignores the composable. For the third button I have marked the text composable of the button as invisibleToUser. Since there is no name associated to the composable Talkback ignores it.

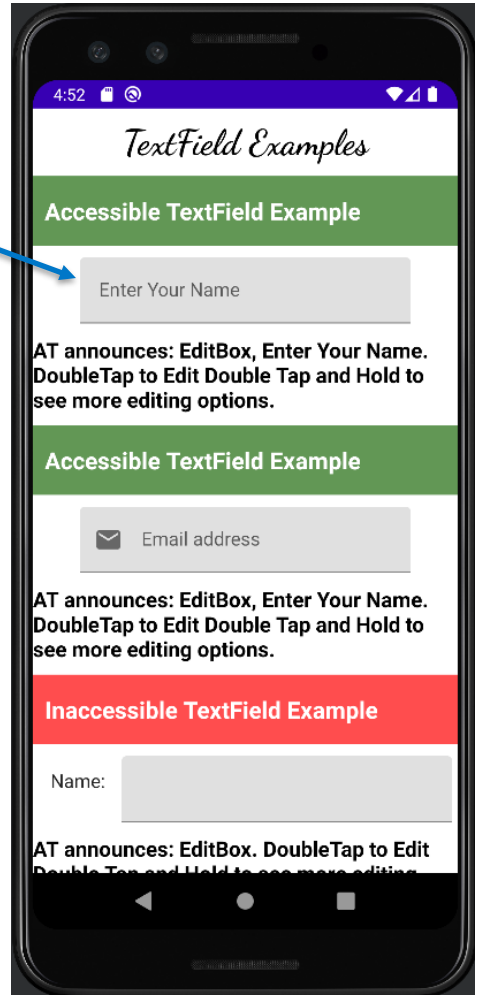
```
Button(
    onClick = { Toast.makeText(context, text: "This is an Inaccessible Button!", Toast.LENGTH_SHORT).show()
    modifier = Modifier.constrainAs(example2) { this: ConstrainScope
        top.linkTo(divider2.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) { this: RowScope
    Text(
        text = "Inaccessible Button",
        modifier = Modifier.semantics { this.invisibleToUser() }
    )
}
```



TextField/EditText

The first TextField is accessible because talkback announces: “EditBox, Enter Your Name. Double tap to Edit, Double Tap and Hold for more editing options”.
Again the name, role and action of are announced by the Talkback.
The text “Enter your name” is associated to the TextField as a label.
This how it looks like in code:

```
val textStateTextField = remember { mutableStateOf(TextFieldValue()) }
TextField(
    value = textStateTextField.value,
    onValueChange = { textStateTextField.value = it },
    label = { Text(text = "Enter Your Name") },
    modifier = Modifier.constrainAs(example1) { this: ConstrainScope
        top.linkTo(divider1.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
)
```

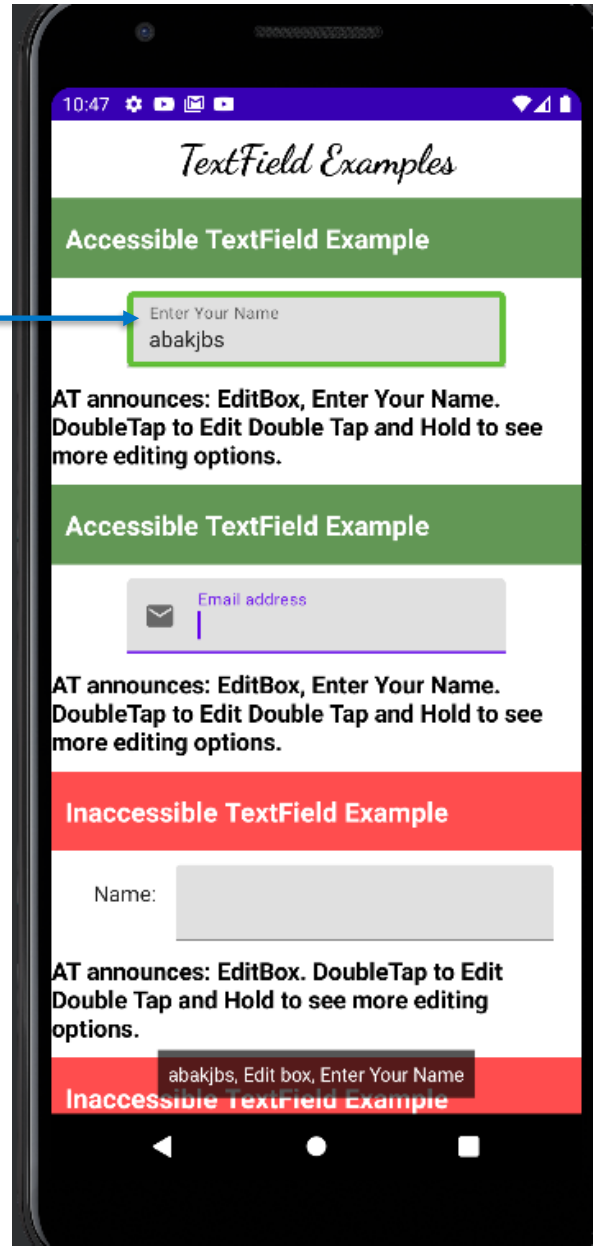


The “onClick” defines the action associated to the button and Actions are part of semantics similar to role.

Labels in TextField/EditText

Accessibility guidelines require labels to persist even when there is text entered into the EditText. Compose does a pretty good job with keeping the label intact all the time.

```
val textStateTextField = remember { mutableStateOf(TextFieldValue()) }
TextField(
    value = textStateTextField.value,
    onValueChange = { textStateTextField.value = it },
    label = { Text(text = "Enter Your Name") },
    modifier = Modifier.constrainAs(example1) {  this: ConstrainScope
        top.linkTo(divider1.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
)
```

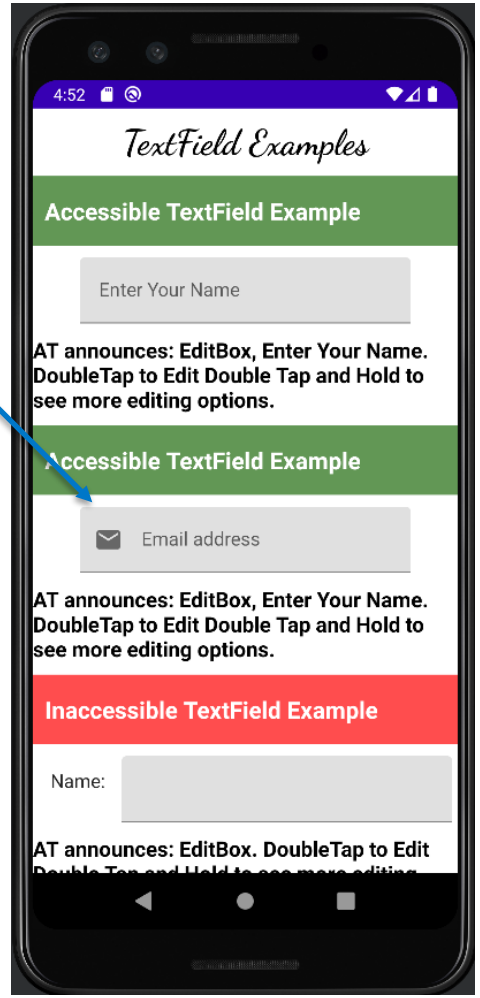


TextField/EditText

The second TextField is accessible because talkback announces: “EditBox, Email Address. Double tap to Edit, Double Tap and Hold for more editing options”.
Again the name, role and action of are announced by the Talkback.
The text “Enter your name” is associated to the TextField as a label.
This how it looks like in code:

```
var text by remember { mutableStateOf(TextFieldValue( text: "")) }
TextField(
    value = text,
    leadingIcon = { Icon(imageVector = Icons.Default.Email, contentDescription = "") },
    //trailingIcon = { Icon(imageVector = Icons.Default.Add, contentDescription = null) },
    onValueChange = { it: TextFieldValue
        text = it
    },
    label = { Text(text = "Email address") },
    modifier = Modifier.constrainAs(example4) { this: ConstrainScope
        top.linkTo(divider4.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
)
```

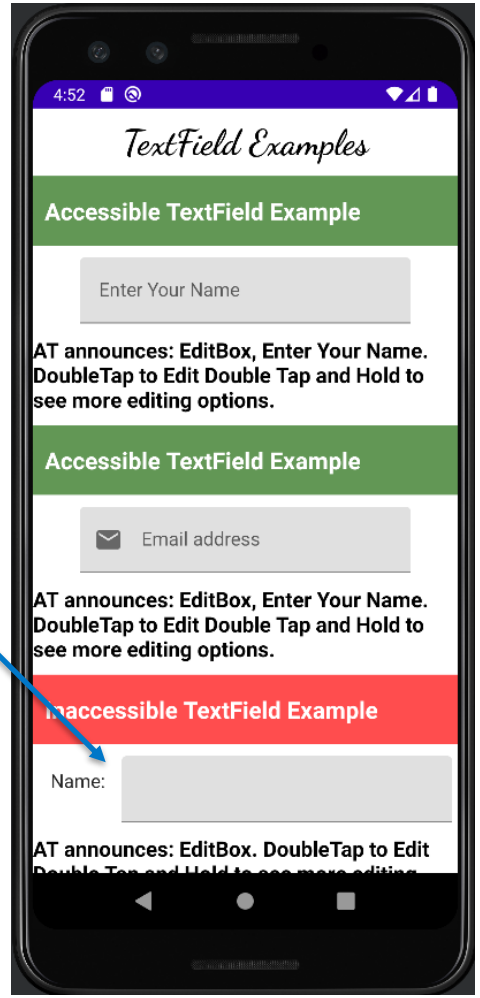
If we provide a content Description to the email Icon then that will also be announced along with the text “Email address”



TextField/EditText

The third TextField is inaccessible because talkback announces: “EditBox, Double tap to Edit, Double Tap and Hold for more editing options”. The label “name” is not part of the TextField. So “Name” is announced separate from the TextField. This how it looks like in code:

```
Row(
    modifier = Modifier.constrainAs(example2) { this: ConstrainScope
        top.linkTo(divider2.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) { this: RowScope

    Text(text = "Name: ", modifier = Modifier.padding(start = 10.dp, end = 10.dp, top = 10.dp))

    val textStateTextField2 = remember { mutableStateOf(TextFieldValue()) }
    TextField(
        value = textStateTextField2.value,
        onValueChange = { textStateTextField2.value = it },
    )
}
```



In the above example we can merge the descendants of the row but that will make Talkback announce: “Name” Role and Action will not be announced which makes the TextField inaccessible.

Merging Descendants

Compose allows us to merge descendants of a composable so that they can be read as one element by Talkback. For example:

```
Row(
    modifier = Modifier.constrainAs(example2) {  this: ConstrainScope
        top.linkTo(divider2.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) {  this: RowScope

    Text(text = "Name: ", modifier = Modifier.padding(start = 10.dp, end = 10.dp, top = 10.dp))

    val textStateTextField2 = remember { mutableStateOf(TextFieldValue()) }
    TextField(
        value = textStateTextField2.value,
        onValueChange = { textStateTextField2.value = it },
    )
}
```

Unmerged descendants

```
Row(
    modifier = Modifier.constrainAs(example2) {  this: ConstrainScope
        top.linkTo(divider2.bottom, margin = 10.dp)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }.semantics(mergeDescendants = true) {  }
) {  this: RowScope

    Text(text = "Name: ", modifier = Modifier.padding(start = 10.dp, end = 10.dp, top = 10.dp))

    val textStateTextField2 = remember { mutableStateOf(TextFieldValue()) }
    TextField(
        value = textStateTextField2.value,
        onValueChange = { textStateTextField2.value = it },
    )
}
```

Merged descendants

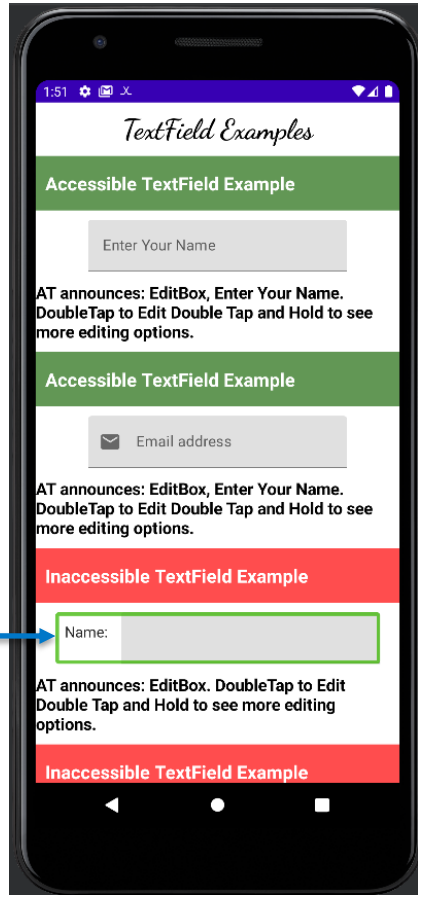
Which means…

Merging Descendants

Unmerged descendants

Talkback focuses the text and the textfield separately.

Talkback groups the text and the Textfield when focusing.



Merged descendants

Be careful while merging descendants, magic happens in certain cases.

Switch

The first Switch is accessible because talkback announces: “On, Get Emails, Switch, Double Tap double tap to toggle.” Creating an accessible Switch in compose is not trivial.

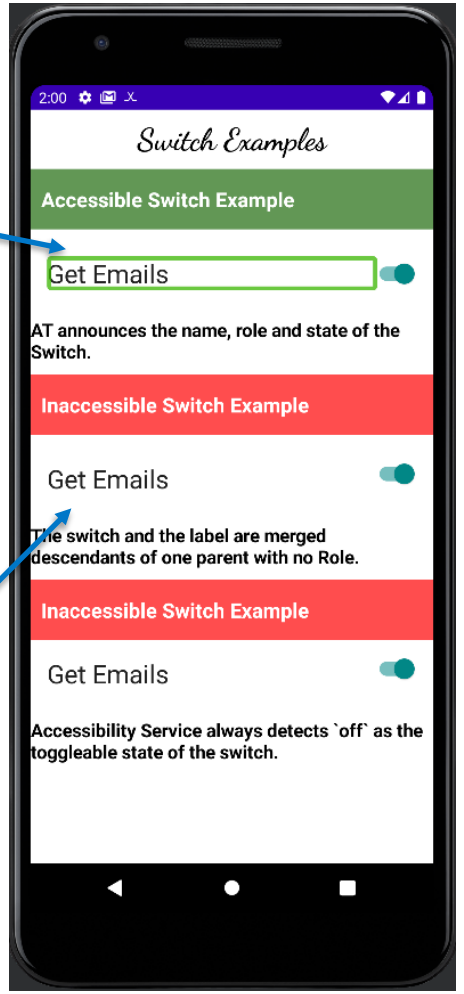
```
val (isSwitchChecked, setSwitchState) = remember { mutableStateOf( value: true) }
Row(
    modifier = Modifier
        .constrainAs(example1) {  this: ConstrainScope
            top.linkTo(divider1.bottom, margin = 10.dp)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
        }
        .semantics(mergeDescendants = true) { }
        .padding(16.dp)
) {  this: RowScope
    Text(
        text = "Get Emails",
        modifier = Modifier
            .toggleable(
                value = isSwitchChecked,
                onValueChange = { setSwitchState(!isSwitchChecked) },
                role = Role.Switch
            )
            .weight( weight: 1f),
        fontSize = 25.sp
    )

    Switch(
        checked = isSwitchChecked,
        onCheckedChange = { setSwitchState(!isSwitchChecked) },
        modifier = Modifier
            .clearAndSetSemantics { }
            .padding(top = 5.dp)
    )
```

Note that the descendants of the Row are merged.

The text is acting like a switch and I have used “toggleable” type modifier to tell the Talkback that by clicking on the text the switch can be toggled.

Since the toggling is happening through the text I have cleared the semantics of the switch.

If the semantics of the switch are not cleared then Talkback focuses on the switch separately.



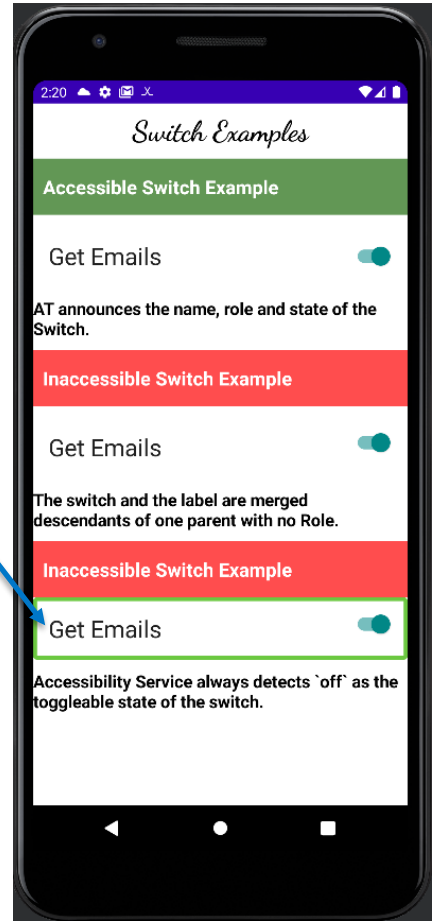
Switch

The third Switch has “toggleable” type modifier set for the parent row which by default add the state “off” to the switch. So the Talkback announces: “On, Get Emails, off, Switch…” notice how “off” state is attached even when the switch is in On state.

```
Row(
    modifier = Modifier.constrainAs(example3) {  this: ConstrainScope
        top.linkTo(divider3.bottom)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
        .semantics(mergeDescendants = true) { }
        .padding(16.dp)
        .toggleable(
            value = isSwitchChecked2,
            role = Role.Switch,
            onValueChange = { setSwitchState2(!isSwitchChecked2) }
        )
) {  this: RowScope
    Text(
        text = "Get Emails",
        modifier = Modifier.weight( weight: 1f),
        fontSize = 25.sp
    )
    Switch(
        checked = isSwitchChecked2,
        onCheckedChange = { setSwitchState2(!isSwitchChecked2) },
        modifier = Modifier.clearAndSetSemantics {  }
    )
}
```

Note that the descendants of the Row are merged.

The Row is acting like a switch and I have used “toggleable” type modifier to tell the Talkback that by clicking on the text the switch can be toggled.



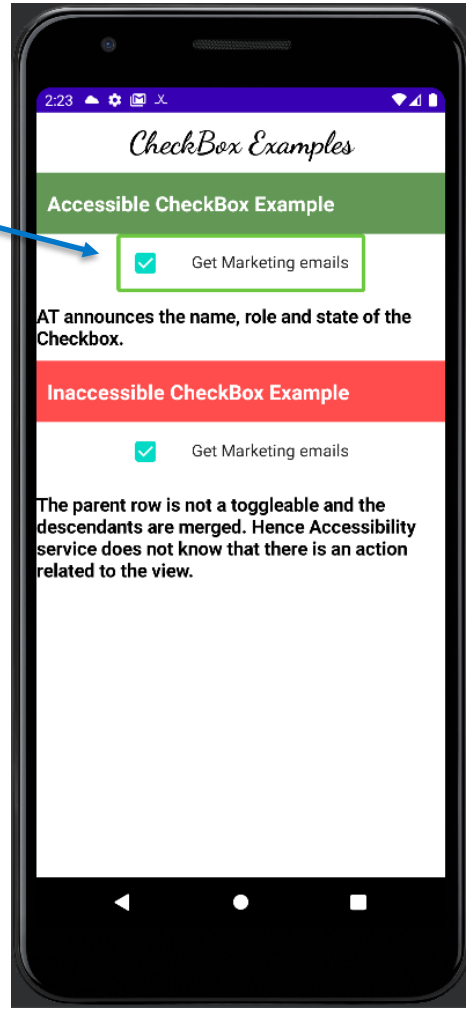
Checkbox

Creating an accessible checkbox in compose is pretty easy. The first checkbox in screenshot announces: “checked, marketing emails, Checkbox, Double tap to toggle.”. So all three name, state and action are announced by the

```
val (checkedState, setCheckBoxState) = remember { mutableStateOf( value: true) }

Row(modifier = Modifier
    .toggleable(
        value = checkedState,
        enabled = true,
        role = Role.Checkbox,
        onValueChange = { setCheckBoxState(!checkedState) }
    )
    .semantics(mergeDescendants = true) { }
    .constrainAs(example1) { this: ConstrainScope
        top.linkTo(divider1.bottom)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) { this: RowScope
    Checkbox(
        checked = checkedState,
        modifier = Modifier.padding(16.dp),
        onCheckedChange = null
    )

    Text(text = "Get Marketing emails", modifier = Modifier.padding(16.dp))
}
```

The Row is acting like a Checkbox and I have used “toggleable” type modifier to tell the Talkback that by clicking on anywhere on the row the Checkbox can be toggled.

Note that the descendants of the Row are merged.



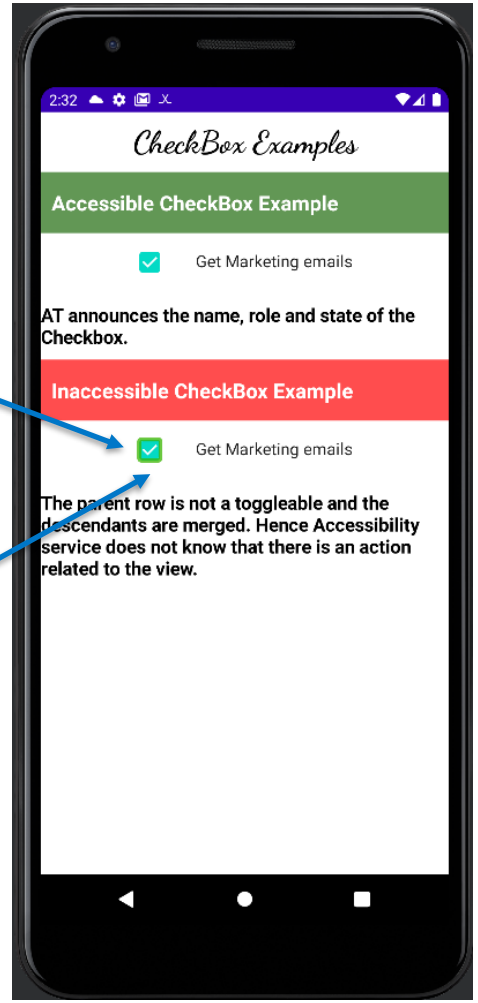
Checkbox

Creating an accessible checkbox in compose is pretty easy. The first checkbox in screenshot announces: “checked, marketing emails, Checkbox, Double tap to toggle.”. So all three name, state and action are announced by the Talkback.

```
val (checkedState1, setCheckBoxState1) = remember { mutableStateOf( value: true) }

Row(modifier = Modifier
    .semantics(mergeDescendants = true) { }
    .constrainAs(example2) {  this: ConstrainScope
        top.linkTo(divider2.bottom)
        start.linkTo(parent.start)
        end.linkTo(parent.end)
    }
) {  this: RowScope
    Checkbox(
        checked = checkedState1,
        modifier = Modifier.padding(16.dp),
        onCheckedChange = { setCheckBoxState1(!checkedState1) }
    )

    Text(text = "Get Marketing emails", modifier = Modifier.padding(16.dp))
}
```

Note that the descendants of the Row are merged.

Talk back first focuses on the whole Row and then on next right swipe it focuses on just the switch.



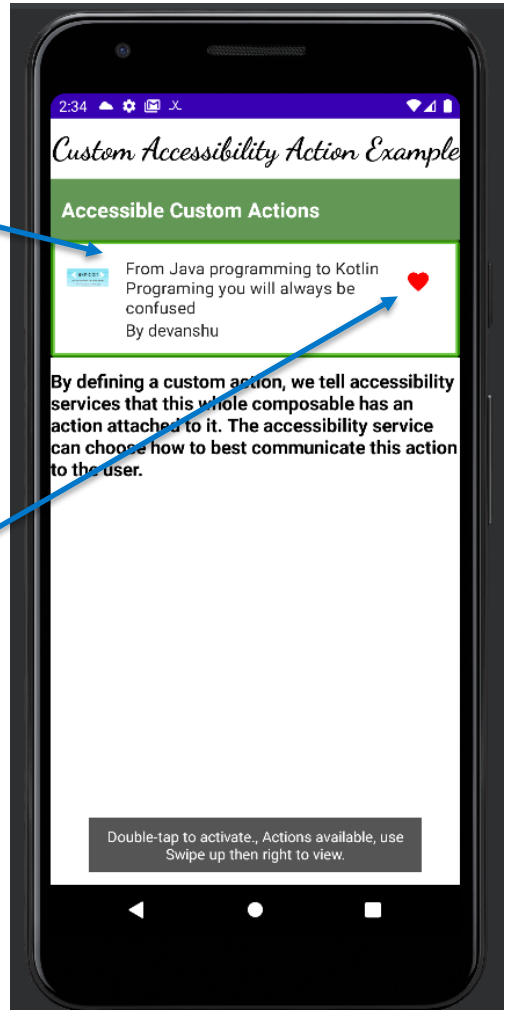
Custom Accessibility Actions

Compose allows us to set custom accessibility actions on composable. Consider a scenario where you want to create a beautiful mark as favorite button on top of an item which actually when clicked takes you to some other screen. So we have to tell Accessibility Service that there are more actions associated to the view and politely ask it to handle those multiple actions.

```
Row(
    modifier = Modifier.border(width = 2.dp, color = Color.Black)
        .clickable(onClick = {
            Toast
                .makeText(context, text: "You have clicked on the item!", Toast.LENGTH_SHORT)
                .show()
        })
        .padding(16.dp)
        .semantics {  this: SemanticsPropertyReceiver
            // Here I am telling the Accessibility Service
            // to treat this Row like a Button
            role = Role.Button
            // By defining a custom action, we tell accessibility services that this whole
            // composable has an action attached to it. The accessibility service can choose
            // how to best communicate this action to the user.
            customActions = listOf(
                CustomAccessibilityAction(
                    label = "selected or unselected book mark",
                    action = {
                        setCheckBoxState(!checkedState)
                        true  ^lambda
                    }
                )
            )
        }
```

Setting custom accessibility action for the row.

Custom accessibility action for the mark as favorite button.

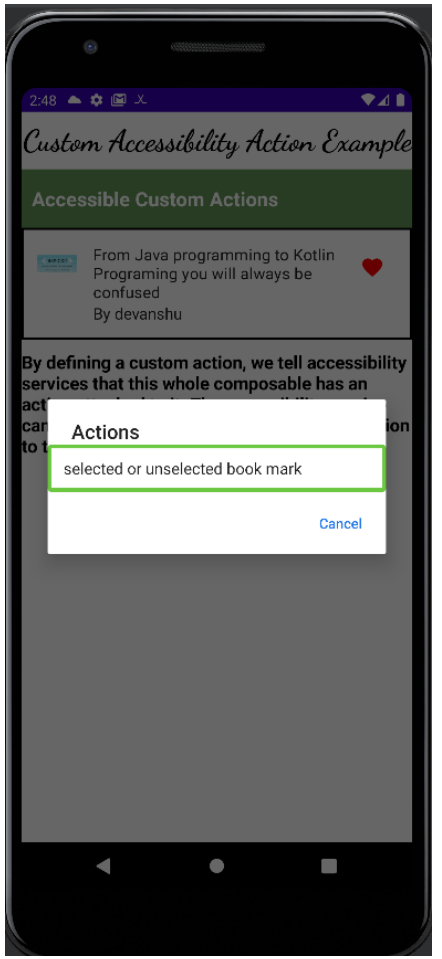


Custom Accessibility Actions

Swiping up and right gives more options to select from.

```
Row(
    modifier = Modifier.border(width = 2.dp, color = Color.Black)
        .clickable(onClick = {
            Toast
                .makeText(context, text: "You have clicked on the item!", Toast.LENGTH_SHORT)
                .show()
        })
        .padding(16.dp)
        .semantics { this: SemanticsPropertyReceiver
            // Here I am telling the Accessibility Service
            // to treat this Row like a Button
            role = Role.Button
            // By defining a custom action, we tell accessibility services that this whole
            // composable has an action attached to it. The accessibility service can choose
            // how to best communicate this action to the user.
            customActions = listOf(
                CustomAccessibilityAction(
                    label = "selected or unselected book mark",
                    action = {
                        setCheckBoxState(!checkedState)
                        true ^lambda
                    }
                )
            )
        }
```

This text is listed as one of the custom actions.



Custom Accessibility Actions with imperative type views

Creating custom accessibility actions in imperative(or xml) type views
is not easy and requires a lot of testing as well as coding efforts.

```
class AccessibilityNodeInfoExt : AccessibilityDelegateCompat() {
  override fun onInitializeAccessibilityNodeInfo(
      host: View?,
      info: AccessibilityNodeInfoCompat?
  ) {
      super.onInitializeAccessibilityNodeInfo(host, info)

      val action = AccessibilityNodeInfoCompat.AccessibilityActionCompat(
          AccessibilityNodeInfoCompat.ACTION_CLICK, host?.context?.getString(R.string.archive_email)
      )

      info?.addAction(action)
  }
}
```

Screen Orientation

Imagine a phone or a tablet mounted on a wheel chair in horizontal orientation.

How to get the Composition Hierarchy?

```
@ExperimentalComposeUiApi
@Test
fun check_ally() {

    composeTestRule.setContent {
        AxeConApp()
    }

    composeTestRule.onNodeWithText( text: "Text Field Examples").assertExists()

    composeTestRule.onRoot(useUnmergedTree = true).printToLog( tag: "TAG-UN-MERGED")
    composeTestRule.onRoot(useUnmergedTree = false).printToLog( tag: "TAG-MERGED")
}
```

To print unmerged tree

To print merged tree

Sample Automated test for compose

How should I test my application?

- Use Accessibility Services like Talkback or Switch Access to test your application.
- Remember, an app’s experience should be identical as a non-AssistiveTechnology user and an AssistiveTechnology user.

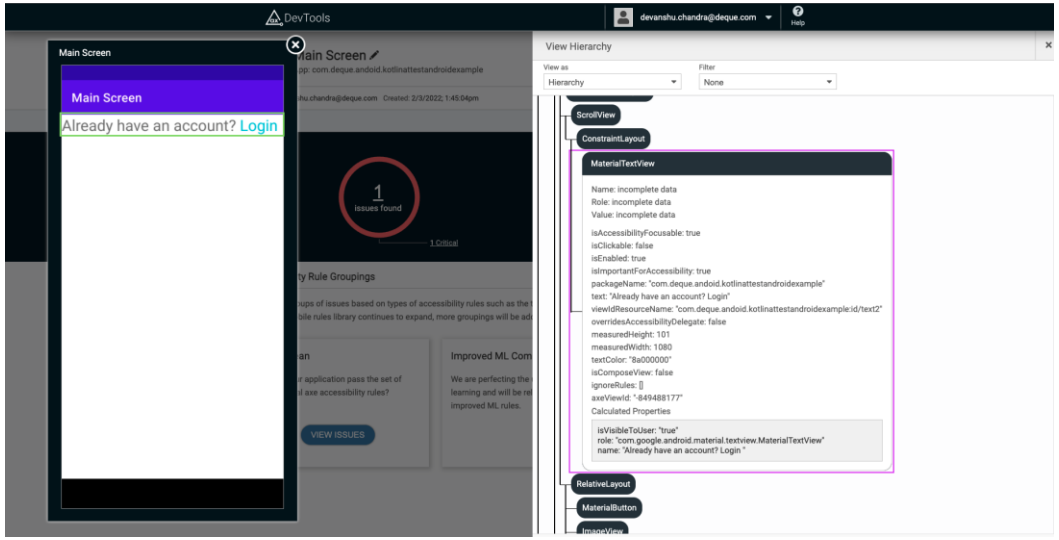
What do we do at Deque?

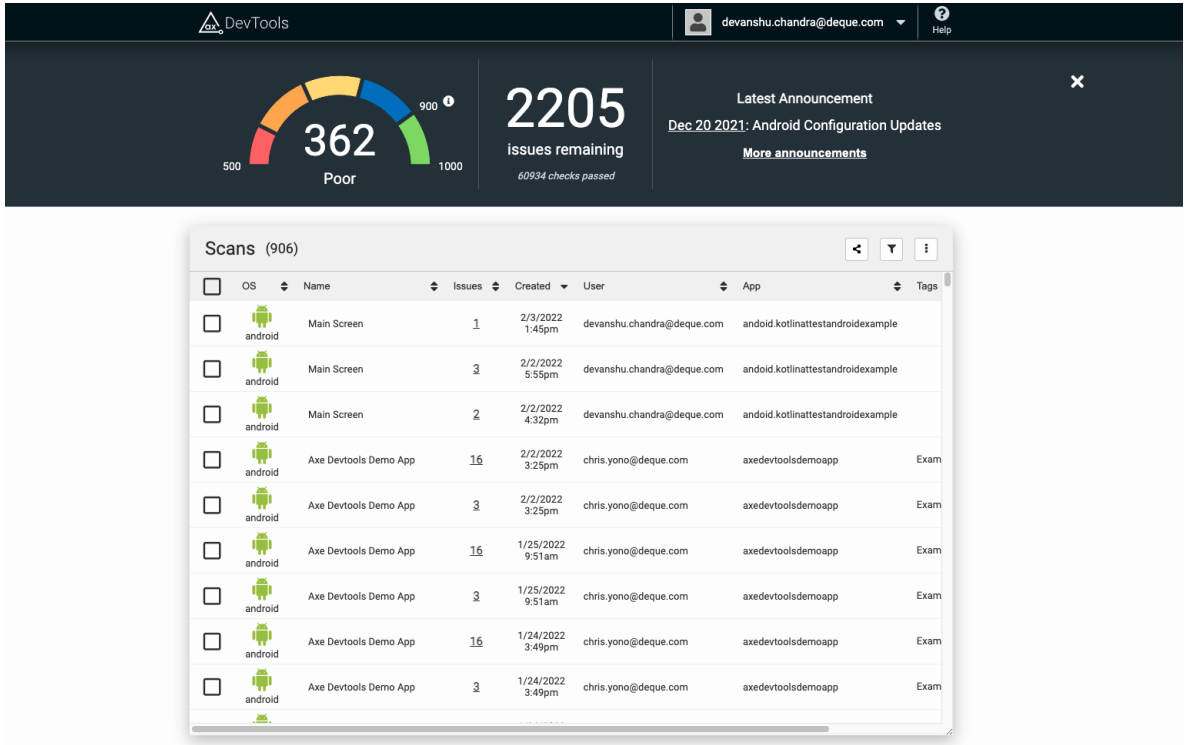
- We develop native Android and iOS libraries which help in detecting accessibility bugs in real time applications.
- Our main focus with the library is to help our customers integrate the library with minimum amount of effort.
- The library can be integrated with any type of automated testing suite be it espresso tests or Appium tests.
- Good News: Our library supports Jetpack compose.
- One can begin testing their application with just one line of code:

```
AxeDevToolsCompose(composeTestRule)
```

What do we do at Deque?

- We also have a pretty intuitive dashboard and documentation which helps in understanding the accessibility issues and directions on how to fix them.





Questions

Connect with me

 chandradevanshu

 devchan4188

 devanshu-chandra-84747441